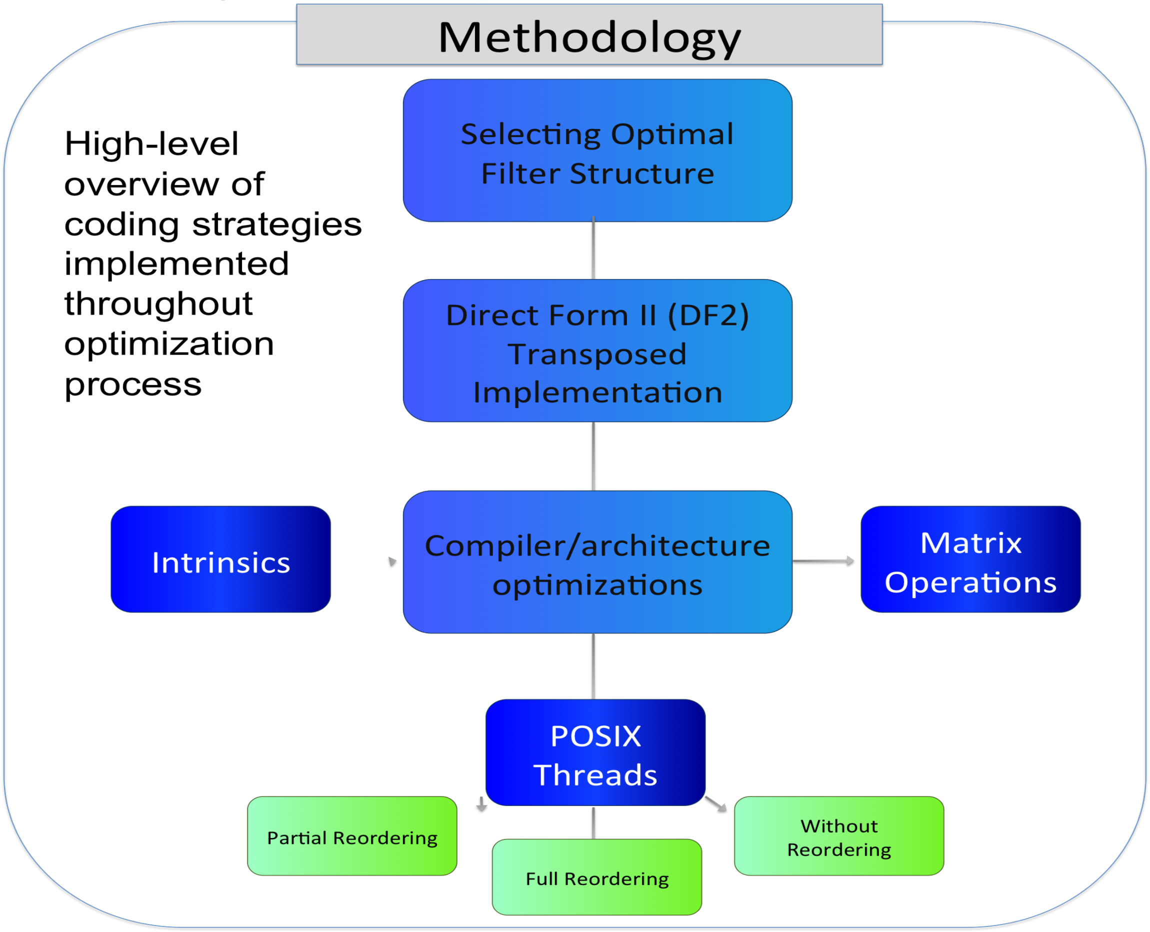
Introduction and Motivation

With the advent of optogenetic tools for neural stimulation and study, the need for enhanced neural recording and processing procedures has grown substantially. One key necessity is to implant and record from many electrodes in the brain at once so as to enhance spatial resolution for a closer analysis of stimuli responses. For optogentic and similar time sensitive experiments, it is quintessential that the data be processed in real time so that appropriate feedback can be delivered. This requires streamlined recording and processing. An open source project under the name "Open Ephys" seeks to accomplish this by creating a light and flexible platform for acquiring electrophysiological data, controlling experimental stimuli, and processing recorded activity.

One feature of the project is a GUI for adding filters and scopes to incoming recordings. In its current state, when attempting to collect data from a moderate number of channels (greater than 32), CPU usage spikes and a slow down in processing the input becomes significant. Our goal is to enhance the current filter library to accommodate real-time filtering for a large number of channels such as 256. We designed an optimized filter in C++ by exploring various forms of difference equations, taking advantage of compiler optimizations, exploiting the computer architecture and multi-threading technologies, in addition to experimenting with matrix implementations.

Overview of the Approach

To begin improving the existing filter bank in Open Ephys, we chose to design and implement a 4-pole Butterworth bandpass filter in C++. In our optimization process, we were sure to write our code in a fashion that easily lent itself to implementing other filter specifications without requiring an overhaul of the code. Our optimization process first required us to select an appropriate filter structure for implementation. We were then free to experiment with compiler optimization flags, intrinsic instruction sets, multithreading, and matrix implementations.


Methodology
High-level overview of coding strategies implemented throughout optimization process
Selecting Optimal Filter Structure
Direct Form II (DF2) Transposed Implementation
Intrinsics
Compiler/architecture optimizations
Matrix Operations
POSIX Threads
Partial Reordering
Full Reordering
Without Reordering

Outline of our approach to this project.

Filter Selection and Construction

Filter Form Selection

Open Ephys Requirements

The vast majority of neural signals require some form of bandpass filtering to be useful. In general, the DC offset must be removed from signals, along with the high-frequency noise, leaving the actual neural data of interest. The Open Ephys GUI employs a simple 4-pole Butterworth bandpass filter for its filtering. An IIR filter is used because the group delay in the pass-band is lower than that of a FIR filter. For garnering a fast response time, keeping the group delay at a minimum is important. While it is impossible for IIR filters to have a linear phase response, the pass-band response has a relatively linear phase. Creating and optimizing a digital IIR Butterworth bandpass filter that operates on a large number of channels will provide the Open Ephys project with a fast filtering solution for the GUI.

Filter Setup

Reasonable digital low-pass and high-pass IIR filters can be designed using only two zeros and two poles. This is commonly known as the digital biquad filter. The difference equation for a digital biquad filter is given below:

Equation:

$$y\left[n\right]=b_{2}x\left[n\right]+b_{1}x\left[n-1\right]+b_{2}x\left[n-2\right]-a_{2}y\left[n-2\right]-a_{1}y\left[n-1\right]$$

And corresponding transfer function:

Equation:

$$H\left(z\right)=\frac{b_{0}+b_{1}z^{-1}+b_{2}z^{-2}}{1+a_{1}z^{-1}+a_{2}z^{-2}}$$

To create a bandpass filter with tunable low and high cutoff frequencies, we can simply cascade a low-pass and a high-pass filter. Cascading the filters is equivalent to multiplying the frequency responses of the two digital biquads, which creates a transfer function with 4 poles and 4 zeros:
Equation:

$$H\left(z\right) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3} + b_4 z^{-4}}{1 + a_1 z^{-1} + a_2 z^{-2} + b_3 z^{-3} + b_4 z^{-4}}$$

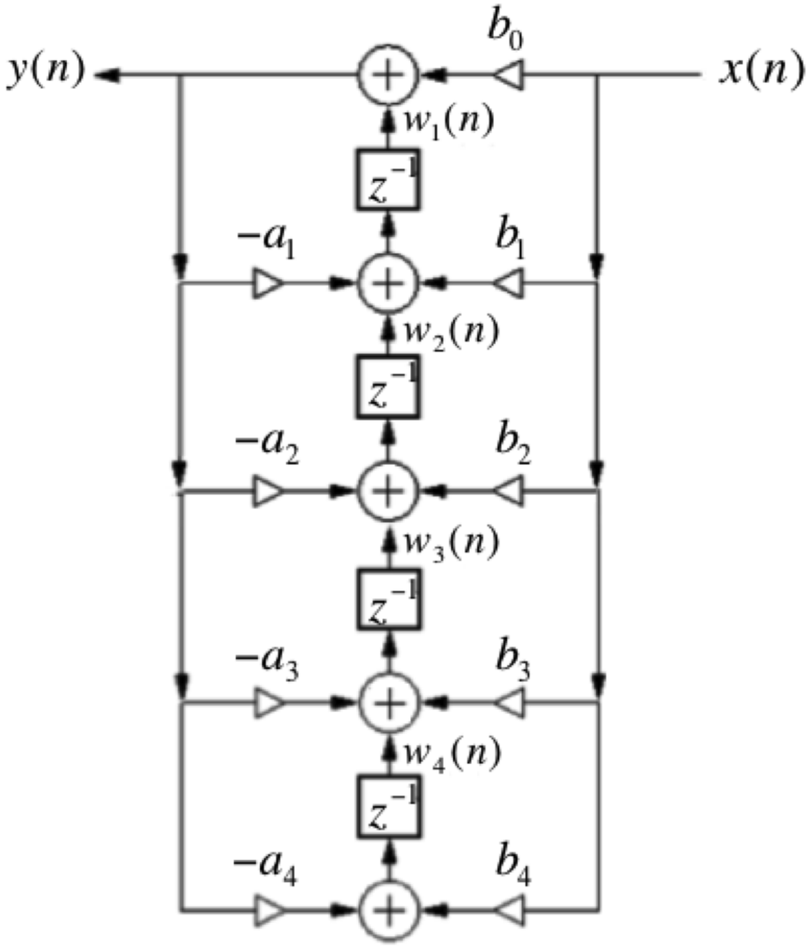
To generate the filter coefficients for a typical bandpass filter for neuroscience experiments, we used MATLAB's Butterworth filter generator with a passband frequency range from 0.1 to 300 Hz and an order of 4:

```
[b,a] = butter(2,[0.1 300]/25000)
```

Direct Form II Transposed Filter

Implementing the Filter

Transpose-form filters are derived as a result of the flow-graph reversal theorem. The filter assumes 0 for all initial conditions. The figure below shows the block diagram for a digital biquad filter. For the bandpass filter that we are building, we need to add two additional transposed stages to account for the second biquad. The final transposed filter structure is shown in Figure 1 below.



Direct Form 2 Transposed 4-Pole IIR Filter

We defined the transposed filter mathematically by a series of equations. Intermediate variables were assigned to the central stages.

Equation:

$$y\left[n\right]=b_0x\left[n\right]+w_1\left[n-1\right]$$

Equation:

$$w_1\left[n\right]=b_1x\left[n\right]-a_1y\left[n\right]+w_2\left[n-1\right]$$

Equation:

$$w_2\left[n\right] = b_2x\left[n\right] - a_2y\left[n\right] + w_3\left[n-1\right]$$

Equation:

$$w_3\left[n\right] = b_3x\left[n\right] - a_3y\left[n\right] + w_4\left[n-1\right]$$

Equation:

$$w_4\left[n\right] = b_4x\left[n\right] - a_4y\left[n\right]$$

This system of difference equations appears to be very different from the difference equation we require, but quick substitution of the intermediate w functions shows that the difference equation is no different from a cascade of digital biquad filters.

Motivation for Use

The Direct Form II Transposed filter is fundamentally the same as other filter configurations. However, it differs because it requires fewer delay units and therefore fewer memory accesses. We can henceforth expect that this filter transposition applies to this implementation as it achieves high function throughput without added register pipelining. Utilizing a Direct Form I filter implementation would require us to constantly keep track of 4 previous inputs and 4 previous outputs. This form allows us to keep track of only 4 intermediate values, and use the current inputs and outputs for calculations.

Compiler Optimization Flags and Intrinsics

Compiler Optimization Flags

We began our optimization process by exploring the performance enhancements using the compiler with optimization flags. We used the GNU Compiler Collection (GCC) and were able to use the O3 optimization flag. O3 is less focused on compiling the code in a way that yields easy debugging, but enables some very important built in optimizations, including inline function expansion and loop unrolling.

- **Function inlining** tells the compiler to insert the entire body of the function where the function is called, instead of creating the code to call the function.
- **Loop unrolling** eliminates the instructions that control the loop and rewrites the loop as a repeated sequence of similar code, optimizing the program's execution speed at the expense of its binary size.

SSE3 Intrinsic Instruction Set

Utilizing different instruction sets provides another opportunity for significant speed gains. We utilized the Intel Streaming SIMD (single-instruction, multiple-data) Extensions 3 technology (SSE3) to improve the the efficiency of the floating-point operations of addition and multiplication. SSE3 instructions are designed for parallel mathematical operations: each processor core contains eight 128-bit SSE registers, which are capable of storing up to 4 floating-point numbers. SSE operations basically perform operations on all 4 floats at the same time, providing a considerable increase in the computation speed for banks of numbers. Because digital filtering is essentially a large number of floating-point multiplications and additions, we felt that SSE would be a perfect addition to the project.

The SSE3 instruction set was implemented using two different methods:

- **Compiler Optimized:** By including the header file `<xmmintrin.h>` and compiling with the O3 optimization flag, the

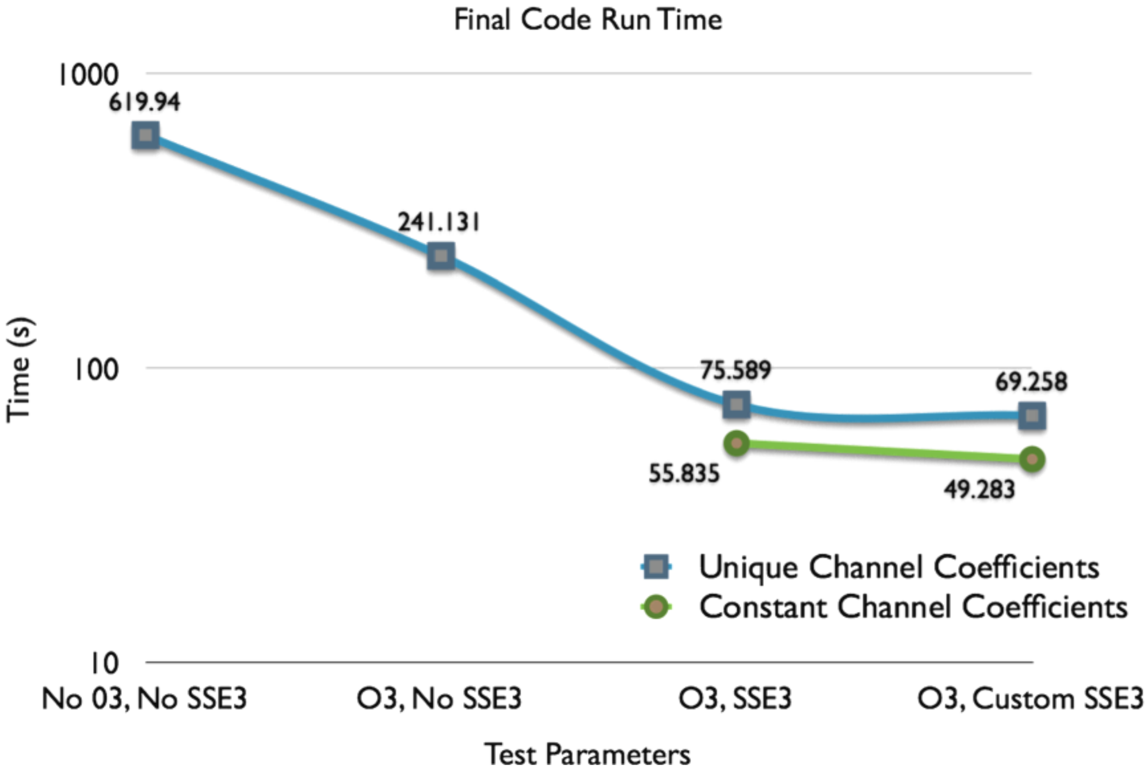
GCC compiler will automatically apply SSE instructions to speed up instrcutions where it deems fit.

- **User-Defined SSE Instructions with Intrinsic Functions:** The SSE header file also grants access to intrinsic functions, which allows us to specifically indicate to the compiler how SSE should be used in our program. We wrote a custom SSE implementation of our filter processing code. Each SSE operation performs calculations on 4 channels simultaneously.

Comparison of Initial Optimizations

The following results were generated on an AMD A6-3400M quad-core processor. We filtered 256 channels with 600,000 time samples. We selected a large number of samples to process to prevent the processor from putting the data in low-level cache, which emulates the behavior of real-time data. The entire program was cycled 100 times to provide temporal resolution of the results, which lets us easily see changes in performance.

Comparison of Initial Optimizations



The data line with unique channel coefficients had a and b filtering coefficients in vectors. The one with constant channel coefficients had fixed coefficients for all channels, which allowed for a quicker run time.

Adding O3 optimization resulted in a speed increase of about 2 binary orders of magnitude. Adding SSE optimizations yielded a speed increase by a factor of more than 3. Utilizing compiler optimization and specialized instruction sets provided a major boost in our filter bank's performance.

Note that we performed tests with filter coefficients uniquely defined for each channel, and also with filter coefficients held the same for all channels. Using the same coefficients for all channels yielded significant speed gains. Most filter banks for neural signals will perform the same bandpass filtering on all channels, so this is an acceptable change for optimization.

Implementation with POSIX Threads
This module contains details of using POSIX threads to attempt to optimize a multi-channel filterbank implementation.

Motivation for POSIX Threads

The popular POSIX (Portable Operating System Interface) Thread API provides a multi-platform interface for creating multi-threaded applications on a variety of UNIX platforms. Multi-threaded applications split the processing load across multiple cores in a computer. Most modern computers (including our test machine) contain CPUs with 2,4, or 8 individual cores, all of which can process data concurrently. Applications can be easily made parallel when significant portions of the data being operated on are independent of any other portion of data.

We recognized that p-threads presented a significant opportunity to improve the efficiency of our filter construction because each channel's data stream is completely independent of any other channel's data. Therefore, we sought to split the processing load for all the channels to multiple threads, all of which can run concurrently.

Implementation 1: Compiler Optimizations Only

The first p-thread implementation is very similar to previous single-threaded implementations. The parameters and data structures are exactly the same as before. The filter coefficients were held constant for all channels (for speed). The primary difference here is that each thread is responsible for filtering a portion of the channels. For example, with 256 channels and 4 p-threads, use the following equal division of processing load:

- Thread 0: process channels 0 - 63
- Thread 1: process channels 64 - 127
- Thread 2: process channels 128 - 191
- Thread 3: process channels 192 - 255

The code was designed to handle arbitrary numbers of p-threads with the pre-condition that the number of threads evenly divides the number of channels. We did our test runs using the standard configuration (256 channels, 600,000 data points, 100 cycles) with 1, 2, 4, 8, 16 and 32 p-threads. A control run with all operations performed on the main thread was also executed. Results for all runs are shown in Table 1.

Number of Threads	Runtime (s)
Control (main thread)	48.075
1	64.302
2	96.755
4	123.931
8	67.629
16	141.329
32	121.134

Implementation 2: Custom SSE3 Intrinsic Optimizations

The results we obtained from implementation results appear promising, but note that all runs were slower than the fastest single-threaded implementation. We decided to apply the intrinsic operations from the SSE3 instruction set that we developed in the previous section to the p-thread applications. Note that for SSE3 to work with p-threads, the pre-condition for the number of p-threads is modified. SSE3 only operates on batches of 4 floats at a time, so the number of channels that each thread operates on must be divisible by 4. In essence:

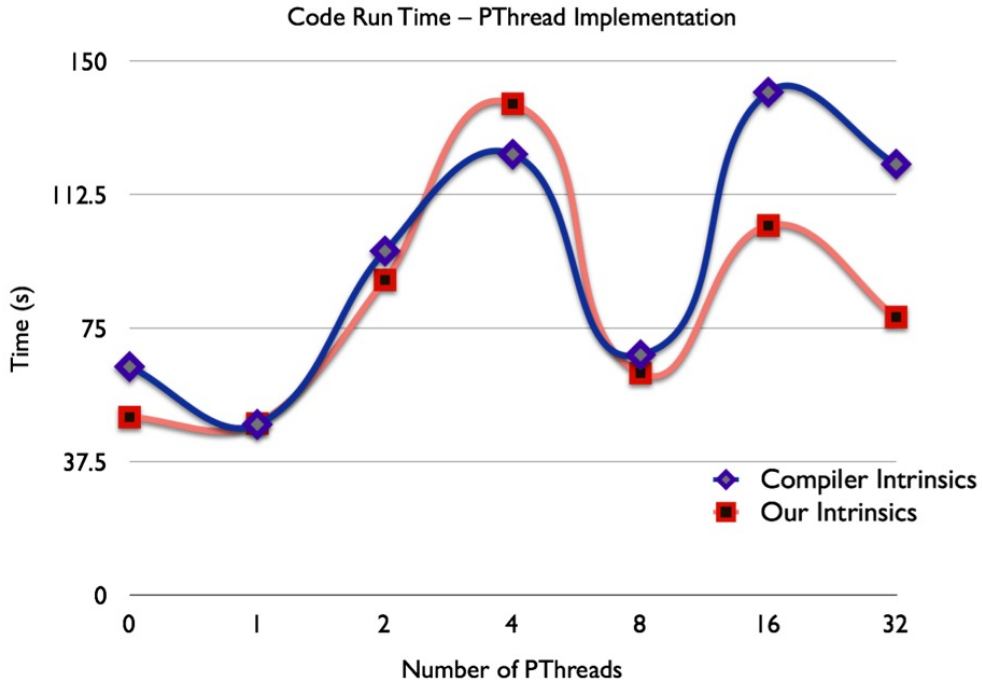
Equation:

$$\frac{number\ of\ channels}{number\ of\ threads} * \frac{1}{4} \in \mathbb{Z}$$

With the same standard run configuration, this pre-condition still supported test runs with 1, 2, 4, 8, 16 and 32 p-threads, along with a control run with execution on the main thread. The results of all runs are shown in Table 2.

Number of Threads	Runtime (s)
Control (main thread)	48.333
1	50.109
2	88.632
4	138.090
8	62.481
16	103.901
32	78.219

Analysis of Results



Note from the figure that the p-thread speed takes much longer for low (between 1 and 4) and high (greater than 8) numbers of p-threads. Marginally decent performance occurs with 8 p-threads on our benchmark computer, which yielded a result of 67.6 seconds using compiler optimizations, and 62.5 seconds using our SSE3 code. Note that the run time for the single-threaded implementation runs at around 48 seconds.

The behavior here definitely does not seem intuitive. With higher processor utilization, the multi-threaded runs take longer than their single-threaded counterparts. After some thought, we concluded that three events were occurring:

- **Cache Missing:** Each CPU core contains a cache which allows processors faster access to data. When a piece of data is requested, it is pulled from memory into cache in a chunk, called a “cache line”. Subsequent accesses to memory nearby the original access will be pulled from the cache, which is much faster. A data request which cannot be found in cache is called a cache “miss,” and a request that is found in cache is called a cache “hit.” When there are very few p-threads, each thread operates on a large portion of memory. Also, each channel's intermediate variables (in the code are in arrays named w1, w2, w3 and w4) are stored in different arrays, which spreads them out in memory.

- **Cache Poisoning:** Each thread runs its own data so one thread will never copy over another thread's data. However, one thread will be operating on data that is nearby in memory to another thread's data. Each thread (running on each core) will be using its own cache. When a thread updates memory that is located in another thread's cache, it will inform the other thread that it is updating the data, and the other cache will mark that information as “dirty.” The subsequent read will occur from memory. We believe with really large numbers of p-threads (greater than 8), the threads operate on smaller chunks of memory that are closer together, resulting in a higher number of cache poisonings.
- **CPU Utilization and P-thread Overhead:** The benchmark computer was equipped with a quad-core processor. This would normally imply that 4 p-threads is optimal, but advances in CPU architecture which are beyond the scope of this research project show that 2 p-threads run more effectively on each core. Also, as the number of p-threads increases, the overhead required to manage them grows, and the actual effectiveness diminishes.

Taking these issues into accounts, we focused our efforts into reorganizing the data structures used by the filter bank.

Data Reordering to Improve POSIX Thread Implementation

This module analyzes the effectiveness of reording data structures in a filter bank implementation to achieve higher efficiency.

This module is a continuation of the previous module on POSIX thread implementations.

Implementation 3: Partial Reordering

An area for improvement over the previous p-thread implementations is to reorder some of the arrays that the filter uses to store data. We started by restructuring the output vector. We split the output vector into a two-dimensional array giving each thread its own output vector. This separates each thread's data in memory, cutting down on cache poisonings. Note that this does change the structure of the final output, but this change is acceptable given that the Open Ephys GUI can be threaded to read in the filter output from several vectors at once.

We ran the same sequence of tests as before using this modified implementations and observed the following results:

Number of Threads	Runtime (s)
Control (main thread)	50.106
1	49.302
2	78.888
4	89.939
8	35.283

16	71.337
32	109.112

Implementation 4: Full Reordering

Another area for significant improvement was to restructure the intermediate data vectors. The vectors were originally designed to store the intermediate filter values of w1, w2, w3 and w4 (see module on Transposed Direct-Form II implementation) in separate arrays. All the threads also shared the arrays, but wrote to different values within the array.

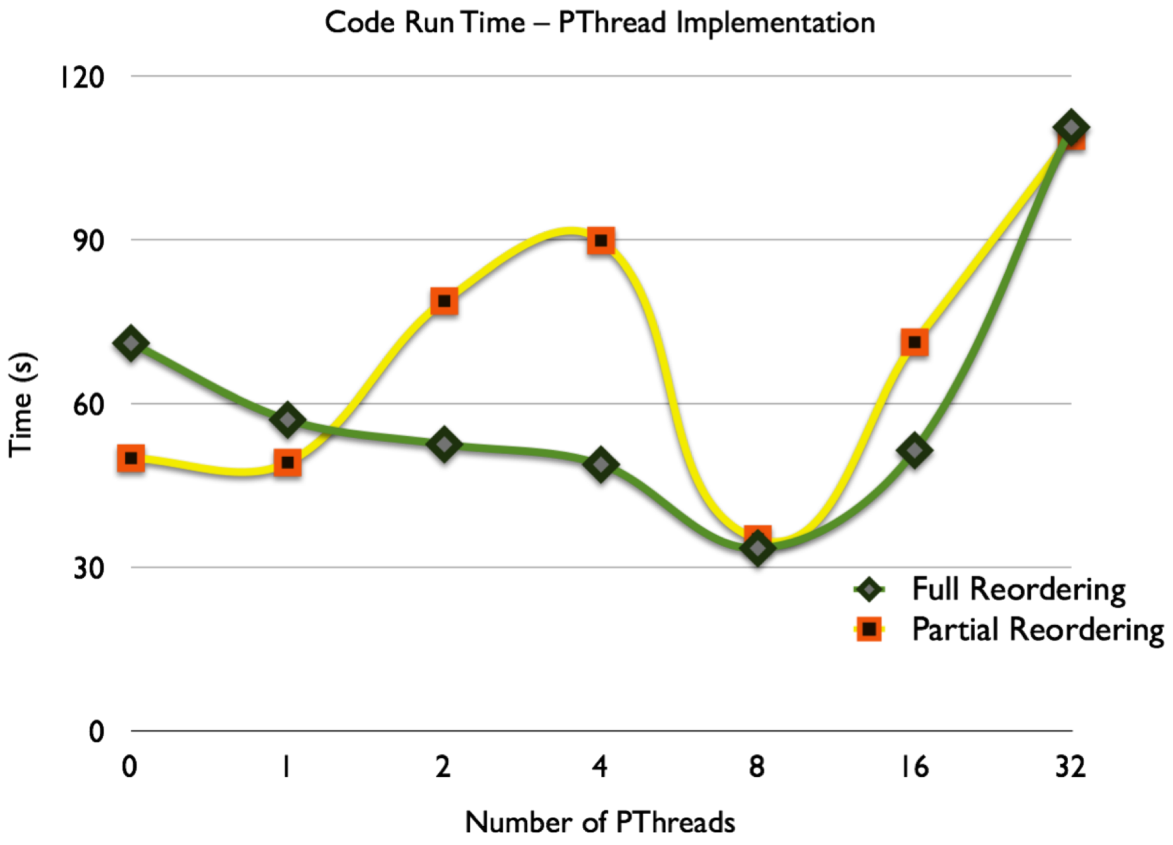
We constructed an alternate structure where all the intermediate filter values of each channel were located in adjacent memory values in a single large vector. Each thread would have its own intermediate value vector for the channels that it was processing. What this does is enable spatial locality for the intermediate values for each channel, which tend to be used all at the same time (temporal locality). The locality of this data ensure that cache hits occur frequently. Splitting the intermediate value vector by thread will help limit cache poisoning.

Using the same sequence of tests to benchmark this implementation, we observed the following results:

Number of Threads	Runtime (s)
Control (main thread)	71.166
1	57.156
2	52.639

4	48.939
8	33.589
16	51.543
32	110.716

Analysis of Results



The obtained results show a very promising improvement, especially after implementing a full reordering scheme. With data reordering, the cache effects of the previous POSIX implementations are significantly reduced. We cannot circumvent p-thread overhead, which indicates why a higher number of p-threads still continues to perform poorly, regardless of data reordering.

With clever data reordering, p-thread implementations of the filter bank can provide significant speed gains over comparable single-threaded methods. The fastest run time obtained by these implementations ran in under 34 seconds with full reordering. This is much faster than the 48 second run time posted by the fastest single-threaded implementation.

Matrix Implementation

Handling multiple channels of data at once naturally yearns for an implementation utilizing matrix operations. While filtering one channel is a purely serial process, we can attempt to compute every channel's filtered output and the intermediate filter terms for a Direct Form-II implementation for the next output in the following two matrix operations.

Equation:

$$\begin{matrix} y_1 \\ \vdots \\ y_N \end{matrix} = b_0 \begin{matrix} x_1 \\ \vdots \\ x_N \end{matrix} + \begin{matrix} w_1^1 \\ \vdots \\ w_N^1 \end{matrix}$$

Equation:

$$\begin{matrix} b_1^1 & -a_1^1 & w_1^1 & \cdots & b_N^1 & -a_N^1 & w_N^1 \\ b_1^2 & -a_1^2 & w_1^2 & \cdots & b_N^2 & -a_N^2 & w_N^2 \\ b_1^3 & -a_1^3 & w_1^3 & \cdots & b_N^3 & -a_N^3 & w_N^3 \\ b_1^4 & -a_1^4 & w_1^4 & \cdots & b_N^4 & -a_N^4 & w_N^4 \end{matrix} \begin{matrix} x_1 \\ y_1 \\ 1 \\ \vdots \\ x_N \\ y_N \\ 1 \end{matrix} = \begin{matrix} \widehat{w}_1^1 \\ \widehat{w}_1^2 \\ \widehat{w}_1^3 \\ \widehat{w}_1^4 \\ \vdots \\ \widehat{w}_N^1 \\ \widehat{w}_N^2 \\ \widehat{w}_N^3 \\ \widehat{w}_N^4 \end{matrix}$$

We implemented the above operations using the Basic Linear Algebra Subprograms (BLAS) Fortran Library due to its reputation as a very fast and stable linear albegra library. While the operations are simple mathematically, the filter implementation was the slowest taking almost 10 minutes to filter 60 million samples. The matrices and vectors are very large so they are not stored in cache resulting in the slow down. Additionally, the

SSE3 intrinsics and O3 compiler optimizations are rendered useless when using the Fortran library.

Here, the matrix operations should be outsourced to the GPU which is optimized for this very action. The GPU is responsible for rendering graphics which regularly involves updating many polyhedrons that really are just matrix operations at the core. However, the GPU would not directly follow the proposed procedure. Instead, the GPU would hand off segments of the matrices to be multiplied to its shaders (modern GPU's have thousands of shaders). Each shader then tackles a very small number of multiplies and adds. This divide and conquer strategy is very fast but requires some involved programming.

Results
This module contains the full compilation of testing results for the filter bank project.

Testing Parameters

All data was collected on a laptop equipped with an AMD A6-3400M “Llano” quad-core processor which supports a clock rate of up to 2.3 GHz.

Test Parameters	
Test Filter	4-pole Butterworth bandpass filter
Input channels	256
Time samples	600,000
Data filter cycles	100
Compiler	GCC

The following results tables show the individual parameters of experiments, averaged run times of the program and an indicator of the real-time processing speed of the program (the formula to generate this figure is shown in the equation below).

Equation:

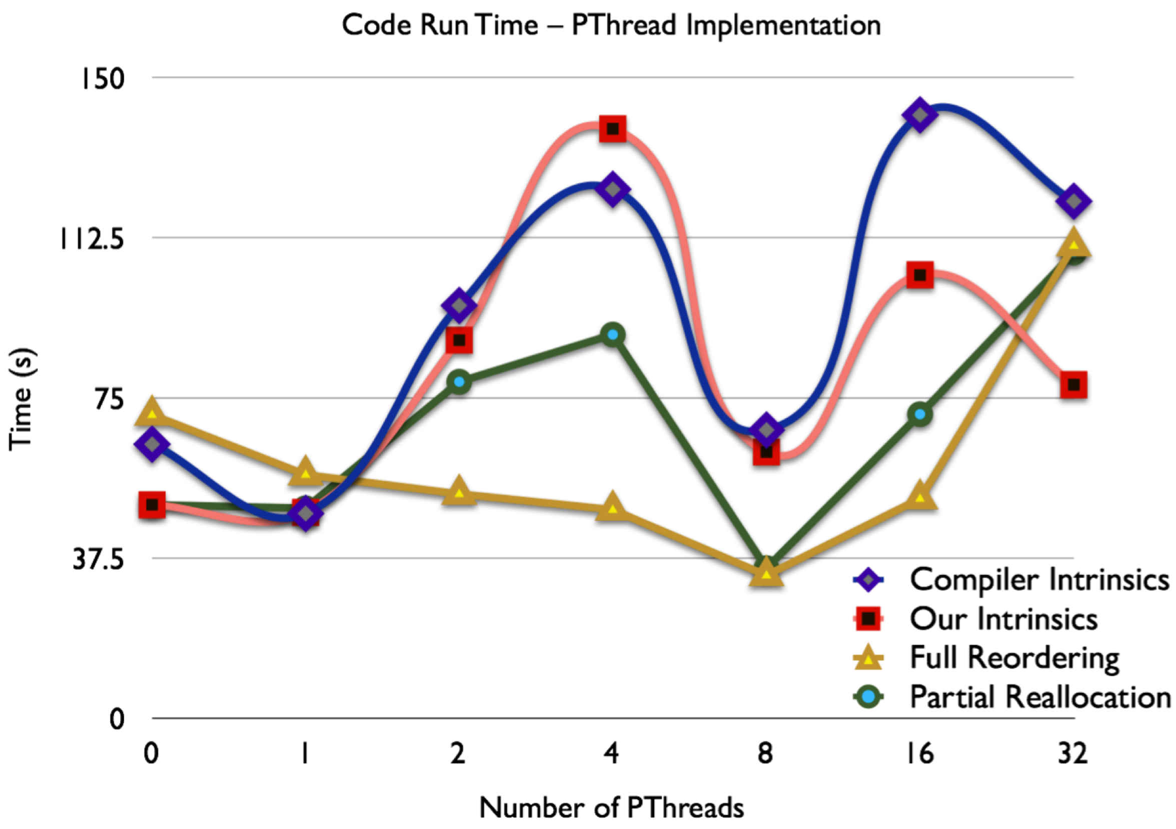
$$\frac{F_s}{\frac{N}{t}} = \frac{F_s t}{N}$$

where F_s = sampling rate of incoming data (samples/sec), N = number of samples processed by filter bank (samples), and t = time to process all samples (sec).

Comparison of Optimizations using Compiler Flags and Intrinsics

Unique Filter Coefficients		
Optimization	Time (sec)	secs/sec (25 KS/s)
None	619.940	0.25831
O3 Compiler	241.131	0.10047
O3 and SSE3	75.589	0.03150
O3 and our SSE3	69.258	0.02886
Constant Filter Coefficients		
O3 and SSE3	55.835	0.02326
O3 and our SSE3	49.271	0.02053

Implementations of POSIX Threads



4 tests under various PThread control conditions demonstrate similar patterns according to the number of PThreads used during code execution. **Compiler Intrinsics**: Implementation with naive data structures relying purely on compiler optimizations of SSE3 intrinsics **Our Intrinsics**: Implementation with naive data structures using our custom SSE3 intrinsics code **Partial Reordering**: Implementation assigning an output vector to each thread utilizing compiler intrinsics **Full Reordering**: Intermediate variables are aligned by channel and separated by thread

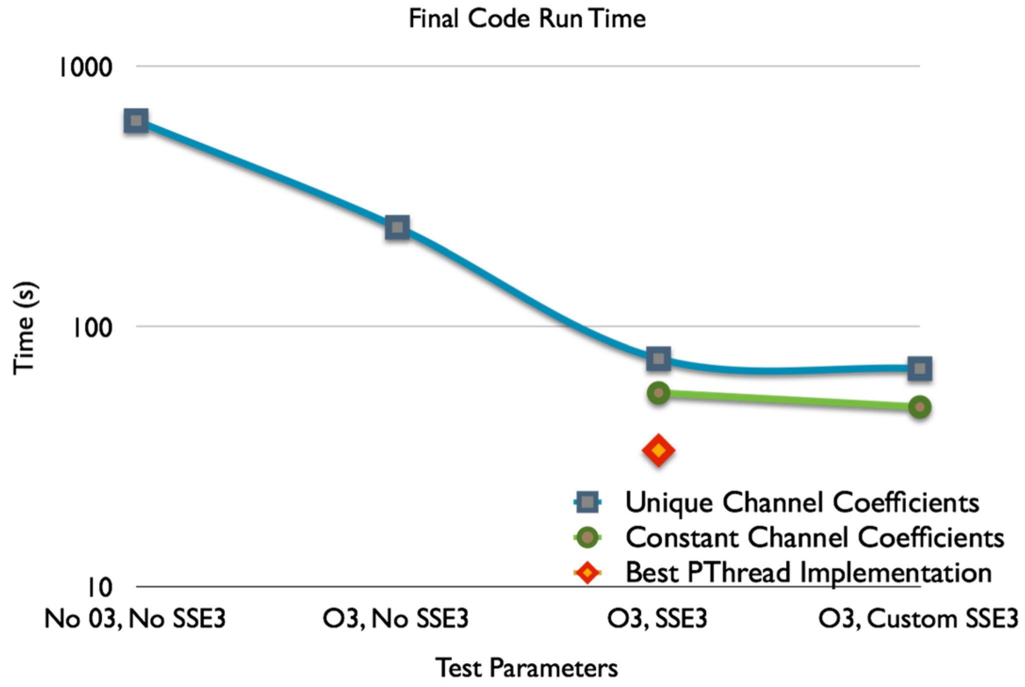
POSIX Threads

Optimization	Time (sec)	secs/sec (25 KS/s)
0 Threads and our SSE3	48.333	0.02014
1 Thread and our SSE3	50.109	0.02088
2 Threads and our SSE3	88.632	0.03693
4 Threads and our SSE3	138.090	0.05754
8 Threads and our SSE3	62.481	0.02603
16 Threads and our SSE3	103.901	0.04329
32 Threads and our SSE3	78.219	0.03259
0 Threads and SSE3	48.075	0.02003
1 Thread and SSE3	64.302	0.02679
2 Threads and SSE3	96.755	0.04031
4 Threads and SSE3	123.931	0.05164
8 Threads and SSE3	67.629	0.02818
16 Threads and SSE3	141.329	0.05889
32 Threads and SSE3	121.134	0.05047

Reordered Output Data

Optimization	Time (sec)	secs/sec (25 KS/s)
0 Threads and SSE3	50.106	0.02088
1 Thread and SSE3	49.302	0.02054
2 Threads and SSE3	78.888	0.02054
4 Threads and SSE3	89.939	0.03747
8 Threads and SSE3	35.283	0.01470
16 Threads and SSE3	71.337	0.02972
32 Threads and SSE3	109.112	0.04546
Reordered Intermediate Variables		
0 Threads and SSE3	71.166	0.02965
1 Thread and SSE3	57.156	0.02382
2 Threads and SSE3	52.639	0.02193
4 Threads and SSE3	48.939	0.02039
8 Threads and SSE3	33.589	0.01400
16 Threads and SSE3	51.543	0.02148
32 Threads and SSE3	110.716	0.04613

Comparison of Initial Optimizations to POSIX Thread Implementation



Both tests show code execution times based on different methods of compiler optimization. Test 1 allowed for unique filter coefficients and test 2 had constant coefficients. Both tests used inefficient data arrangement.

Our optimal filter design incorporated a combination of several of the methods we used to optimize our filter bank implementation. It made use of compiler-level optimization, SSE instructions, and POSIX threads. It processed 60 million samples for each of the 256 channels in 33.589 seconds. Assuming a 25 KHz sampling rate, each second of of data is processed in 14 milliseconds (25 KS/s) and thus acceptable for real-time processing.

Conclusions and Future Work

Conclusions

Our optimal filter implementation utilized the Direct Form II Transposed filter structure with 8 POSIX threads and innovative data structuring with the SSE3 instruction set and the O3 compiler optimization flag to provide the perfect combination between cache hits, CPU utilization, and minimal cache poisoning. Assuming a sampling rate of 25 KHz, we were able to achieve our goal of real-time filtering by processing one second of simulated data in 14 ms (this figure was calculated using the real-time processing speed formula specified in the Results section).

Note that we obtained our results on a single type of computer architecture. We expect that many of our optimization methods, such as SSE and data reordering, will be effective on any architecture. The cache organization of most modern processors are relatively similar to our benchmark computer. Modern CPUs are primarily differentiated in the number of physical cores they contain. This will alter the number of p-threads required for optimal CPU utilization. Machines with more cores will efficiently utilize more p-threads, while machines with fewer cores will utilizes fewer p-threads.

Processing data at this speed is of paramount importance to the Open Ephys project and to neural signal processing in general. The filter bank we have written is designed to provide a generic bandpass filter to all incoming signals. Signals will typically require additional real-time processing afterwards. For example, a good number of neural signal projects need to detect specific phenomena (usually peaks) in real-time and provide impulses to alter the source neuron's behavior.

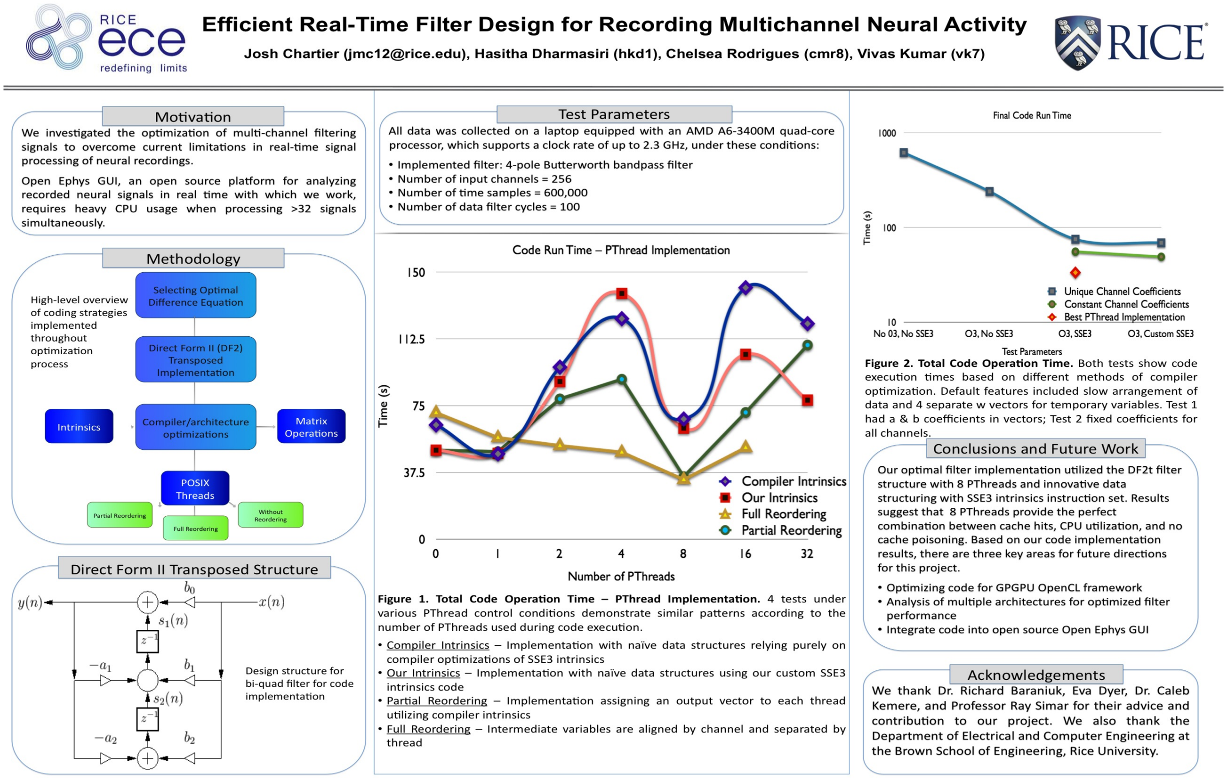
Future Work

This project focused heavily on optimizing code for use on CPU architectures, however, optimizing the code for GPU implementation, especially GPGPU (General-purpose computing on graphics processing units), represents a major area for future performance improvement. GPU units are extremely efficient at manipulating graphics and matrix calculations because of parallelized structuring that more efficiently pipelines instruction sets. As a result, large blocks of data can be simultaneously computed in multichannel processes alleviating the burden of memory allocation and usage that we commonly encountered throughout the development of our project. Another major advantage of optimizing for GPU units is that the OpenCL computing framework upon which modern GPUs are built is a platform independent language. Also, an issue that we encountered in writing our code was that different computer architectures necessitated different optimization techniques, and as a result, code timings for different architectures varied significantly. Solutions made upon the OpenCL framework are extended to various architectures homogeneously, eliminating the need for targeted optimization techniques in addition to the basic compiler optimizations that we built into our code.

The next step in this project is to implement our code into the source code for the Open Ephys GUI. This will allow neuroscientists performing experiments with many electrodes to make use our filter implementation for real-time processing. This will be an invaluable addition to the open source project as the Open Ephys project grows in popularity and as the community tends towards experiments with a large number of electrodes.

Appendix
A poster summarizing our project, as well as acknowledgements.

Poster



Efficient Real-Time Filter Design for Recording Multichannel Neural Activity

Josh Chartier (jmc12@rice.edu), Hasitha Dharmasiri (hkd1), Chelsea Rodrigues (cmr8), Vivas Kumar (vk7)

Motivation

We investigated the optimization of multi-channel filtering signals to overcome current limitations in real-time signal processing of neural recordings.

Open Ephys GUI, an open source platform for analyzing recorded neural signals in real time with which we work, requires heavy CPU usage when processing >32 signals simultaneously.

Methodology

High-level overview of coding strategies implemented throughout optimization process

Selecting Optimal Difference Equation
Direct Form II (DF2) Transposed Implementation
Intrinsics
Compiler/architecture optimizations
Matrix Operations
POSIX Threads
Partial Reordering
Full Reordering
Without Reordering

Direct Form II Transposed Structure

Design structure for bi-quad filter for code implementation

Test Parameters

All data was collected on a laptop equipped with an AMD A6-3400M quad-core processor, which supports a clock rate of up to 2.3 GHz, under these conditions:

- Implemented filter: 4-pole Butterworth bandpass filter
- Number of input channels = 256
- Number of time samples = 600,000
- Number of data filter cycles = 100

Figure 1. Total Code Operation Time – PThread Implementation. 4 tests under various PThread control conditions demonstrate similar patterns according to the number of PThreads used during code execution.

- Compiler Intrinsics – Implementation with naïve data structures relying purely on compiler optimizations of SSE3 intrinsics
- Our Intrinsics – Implementation with naïve data structures using our custom SSE3 intrinsics code
- Partial Reordering – Implementation assigning an output vector to each thread utilizing compiler intrinsics
- Full Reordering – Intermediate variables are aligned by channel and separated by thread

Figure 2. Total Code Operation Time. Both tests show code execution times based on different methods of compiler optimization. Default features included slow arrangement of data and 4 separate w vectors for temporary variables. Test 1 had a & b coefficients in vectors; Test 2 fixed coefficients for all channels.

Conclusions and Future Work

Our optimal filter implementation utilized the DF2t filter structure with 8 PThreads and innovative data structuring with SSE3 intrinsics instruction set. Results suggest that 8 PThreads provide the perfect combination between cache hits, CPU utilization, and no cache poisoning. Based on our code implementation results, there are three key areas for future directions for this project.

- Optimizing code for GPGPU OpenCL framework
- Analysis of multiple architectures for optimized filter performance
- Integrate code into open source Open Ephys GUI

Acknowledgements

We thank Dr. Richard Baraniuk, Eva Dyer, Dr. Caleb Kemere, and Professor Ray Simar for their advice and contribution to our project. We also thank the Department of Electrical and Computer Engineering at the Brown School of Engineering, Rice University.

The poster can be downloaded in PDF form.
https://cnx.org/content/m45318/

The Team

Hasitha Dharmasiri

Hasitha is a junior studying electrical engineering with a focus in signals and systems. He served as the primary code monkey and researcher for p-thread implementation.

Chelsea Rodrigues

Chelsea is a junior studying electrical engineering with a focus in photonics and nanoengineering. She assumed responsibility as the scribe monkey for meetings, learning the upload procedure for Connexions, and researching compiler optimizations.

Josh Chartier
Josh is a junior studying electrical engineering and computational and applied mathematics with a focus in signals and systems. He served as the consultant code monkey and researcher for matrix and SSE3 implementations.

Vivas Kumar
Vivas is a junior studying electrical engineering with a focus in signals and systems. He served as the monkey wrangler and resident expert in all things related to Keynote, Powerpoint, and presenting.

Acknowledgements

We thank Dr. Richard Baraniuk, Eva Dyer, Dr. Caleb Kemere, and Professor Ray Simar for their advice and contribution to our project. We also thank the Department of Electrical and Computer Engineering at the Brown School of Engineering, Rice University.